Meta-programming and Multi-stage Programming for GPGPUs

Marc Baboulin, Joel Falcou, Ian Masliah

Meta-programming and Multi-stage Programming for GPGPUs

Marc Baboulin, Joel Falcou, Ian Masliah

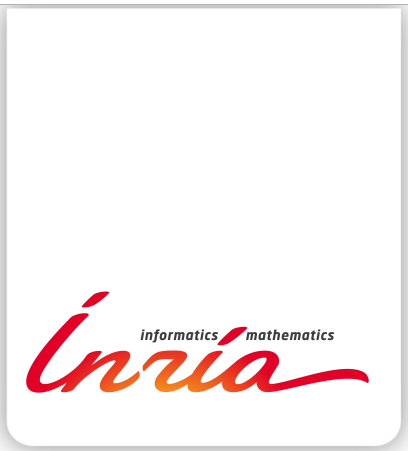
RESEARCH REPORT

N° 8780

Septembre 2015

Project-Team Postale

ISSN 0249-6399 ISRN INRIA/RR--8780--FR+ENG



Meta-programming and Multi-stage Programming for GPGPUs

Marc Baboulin*, Joel Falcou†, Ian Masliah‡

Project-Team Postale

Research Report n° 8780 — Septembre 2015 — 21 pages

Abstract: GPGPUs and other accelerators are becoming a mainstream asset for high-performance computing. Raising the programmability of such hardware is essential to enable users to discover, master and subsequently use accelerators in day-to-day simulations. Furthermore, tools for high-level programming of parallel architectures are becoming a great way to simplify the exploitation of such systems. For this reason, we have extended NT^2 – the Numerical Template Toolbox - a C++ scientific computing library which can generate code for SIMD and multi-threading systems in a transparent way. In this paper, we study how to introduce an accelerator-based programming model into NT^2 to allow developers to reap the benefits of such an architecture from a simple, MATLAB -like code. After a brief description of the NT^2 framework, we explain how our accelerator programming model has been designed and integrated in a pure C++ library. We conclude by showing the applicability and performance of this tool on some practical applications.

Key-words: C++, meta-programming, generic programming, CUDA, Multi-stage

* Inria and Université Paris-Sud, France (marc.baboulin@inria.fr).
† Inria and Université Paris-Sud, France (joel.falcou@lri.fr).
‡ Inria and Université Paris-Sud, France (ian.masliah@lri.fr).

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Méta-programmation et programmation multi-niveaux pour GPGPU

Résumé : La programmation sur les accélérateurs tels que les GPGPU est devenue un atout majeur pour le calcul haute-performance. Une amélioration de la programmabilité de ces composants matériels est essentielle pour permettre aux utilisateurs de maîtriser ces accélérateurs afin d'être capable d'écrire un code plus performant. Les outils permettant une programmation haut niveau des architectures parallèles sont de nos jours un moyen très efficace pour simplifier l'exploitation de telles machines. Dans ce contexte, nous avons étendu la bibliothèque de calcul scientifique nommée NT^2 – the Numerical Template Toolbox - capable de générer des codes pour machines à processeurs vectoriels et multi-coeurs de manière transparente. Dans ce rapport technique, nous étudions des modèles de programmation pour accélérateurs afin de les intégrer dans la bibliothèque NT^2 . L'objectif est de permettre aux développeurs de profiter des capacités des machines parallèles de manière simple avec une interface similaire à MATLAB . Après une brève description de NT^2 , nous expliquons comment nous avons intégré notre modèle de programmation dans une bibliothèque C++. Pour conclure, nous présentons les performances de notre système sur certaines applications du domaine.

Mots-clés : C++ , méta-programmation, programmation générique, CUDA, programmation multi-niveaux

1 Introduction

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. As computing hardware complexity rose with the advent of SIMD, multi-processor, multi-core systems and more recently accelerators like GPUs [22] or Intel Xeon Phi [5], software design methodologies did not undergo the same amount of changes. This complicates the exploitation of such hardware in mainstream applications.

Designing **Domain Specific Languages** (or *DSL*) has been presented as a solution to these issues. As *DSL*s allow solutions to be expressed in their programming idiom with a level of abstraction equivalent to the problem domain, the maintainability and quality of code is increased. One of the most popular examples is MATLAB ™ which provides a large selection of toolboxes that allow a direct expression of high-level algebraic and numerical constructs in a easy-to-use imperative language. In this scope, **Domain Specific Embedded Languages** (or *DSEL*s) [17, 30] are languages implemented inside a general-purpose, host language [8]. without the requirement of a dedicated compiler or interpreter as they are designed as a library-like component [7, 31].

NT^2 – The Numerical Template Toolbox – is such a *DSEL* using C++ template meta-programming [1] to provide a MATLAB -inspired API while supporting a large selection of parallel architectures and keeping a high level of expressiveness [13]. NT^2 was designed to support architectural features like SIMD extensions and multi-core programming [29]. However, the support for accelerators like GPGPUs was limited as GPU kernel compilers where unable to process NT^2 C++ 11 based implementation of C++ based *DSEL* .

In this paper, we present a new extension for NT^2 that takes care of such accelerators, especially CUDA based GPUs through multi-stage programming [12] (or *MSP*) for linear algebra and elementwise problems. *MSP* consists in doing multiple compilation phases allowing for type-safe program generation. Our contributions include:

- A programming model supporting both implicit or explicit data transfers to and from GPGPUs with a simple user interface
- An adaptable strategy to generate CUDA kernel directly from a single C++ source file containing NT^2 statements
- The integration of this kernel generator with existing CUDA libraries like cuBLAS or MAGMA.

The purpose of this system is to provide the user some leeway on how to distribute the data between the host and device through a simple mechanism. As having a perfect cost model for load balancing is very complex to put in place and costly, letting the user provide some insight on data locality is beneficial.

After reviewing the concurrent state of the art software libraries (section 2), we introduce NT^2 , its programming model (section 3) and how it has been adapted to support GPU computation. We then describe the kernel generator process and how it integrates with the existing library (section 4). Finally, we present benchmarks assessing the generated code quality (section 5) and conclude on the future work regarding NT^2 and accelerators.

2 Related Works

Software libraries for GPGPU computing try to simplify the new programming paradigm brought by many-core based systems. The general trend followed in recent years by C++ libraries is to provide a high-level interface through template meta-programming techniques. This goal is reached by providing device containers with architecture-aware generic parallel algorithms and/or a code generation based process. This approach differs from what can be seen with OpenACC [32] or Halide [23] as it is a *DSEL* based approach and does not rely on language extensions or pragmas. We will give a detailed explanation of both type of libraries that support OpenCL/CUDA or both.

Thrust [16] is a header only library providing a similar interface to the C++ Standard Template Library. Its high-level interface is based on meta-programming and traits to provide efficient parallel skeletons that can run on either CPU or GPU. Container locality is expressed through an explicit container declaration limiting the abstraction but allowing for easier transfers between host and device. Locality for functions is defined by default for device vectors and can be extended with tag dispatching. Overall, it is a well rounded utility library which can be combined with CUDA Toolkit libraries such as CUBLAS, CUFFT and NPP. However it lacks of code generation features and does not support OpenCL.

VexCL [10] is an expression template library for OpenCL/CUDA. It provides a high-level generic interface that is suitable for both back-ends with static parameters defined within a *DSEL* for linear algebra. The expression template mechanism allows for code generation by lazy evaluation of vectors and elementary operations within the AST. Similarly to Thrust, it provides STL-like functions on containers that have a defined locality. It is also possible for the user to define custom functions on device that will be dynamically generated for CUDA. However, the transform used for the generation process requires a unique data locality limiting hybrid algorithms.

ViennaCL [25] is also an expression template library for OpenCL/CUDA. This library strictly follows the uBLAS programming interface and STL like algorithms for easier integration with other softwares. It has implementations for BLAS kernels and high-level solvers for sparse and dense computation that provide good performance. Through the mechanism of expression templates it can evaluate basic linear algebra operations with operator overloading. ViennaCL focuses more on OpenCL due to the necessity for separate compilation with CUDA limiting its support through the OpenCL language. It is however possible to generate CUDA code with ViennaCL at runtime.

Boost.Compute [20] is a header only C++ library based on the OpenCL standard. Similar to other libraries, it manages device memory through a designated container. It provides an interesting concept of future for asynchronous copy on the device allowing for more versatility. Boost.Compute also supports closures and adaptable structures for device. Similarly to thrust, it is a well rounded library based on OpenCL to simplify the coding process on accelerators. It however lacks support for numerical analysis and cannot generate CUDA code.

Eigen [15] is a popular library to solve linear systems using expression templates. It should be able to support accelerator code with CUDA by writing Eigen code in a .cu file. It however does not provide any high-level interface or special containers for GPU computing. Eigen does not support OpenCL.

SciPAL [18] is an expression template library with a *DSEL* for dense and sparse linear algebra. It focuses mainly on recognizing gemm calls by parsing the AST and regular operations on containers. Its code generation process for CUDA kernels is done mostly by explicitly writing the code in callable objects. The code generation process is then done at runtime for non-BLAS kernels. SciPAL does not support OpenCL.

Feature	Thrust	VexCL	ViennaCL	Boost.C	NT^2
MATLAB API	–	–	–	–	✓
AST optimization	–	✓	✓	✓	✓
Device Arrays	✓	✓	✓	✓	✓
Cuda code gen	✓	✓	–	–	✓
OpenCL code gen	–	✓	✓	✓	–
parallel skeletons	✓	✓	–	✓	✓
CUBLAS support	✓	–	✓	–	✓
Static code gen	–	–	–	–	✓
dense LA solvers	–	–	✓	–	✓
sparse LA solvers	–	–	✓	–	–

Figure 1: Feature set comparison between NT^2 and similar libraries

In Figure 1, we compare features between NT^2 and the previously described libraries. We did not include SciPAL since the code is not available, and Eigen as it does not have any real support for code generation or a device API.

The purpose of the previously described libraries is usually to provide a wrapper over the C++ language for GPGPU computing. For this reason, the *DSEL*s based on expression templates or `Boost.Proto` are usually lightweight and consist mostly of overloading elementary operations for containers. The code generation phase then ends up doing a dynamic compilation of the CUDA code/OpenCL kernel which adds a significant overhead on small sized or low arithmetic intensity problems.

Furthermore, as mentioned in Section 1, it is not possible to compile a NT^2 source code with nvcc even with the latest version (7.0 RC). To address these issues, we have added a two step compilation in NT^2 using a serialization mechanism based on Boost.Serialization. Before describing this process, we will first detail the execution model for GPGPU computing in NT^2 .

3 NT^2 Execution Model

NT^2 [13] is a numerical computing C++ library implementing a subset of the MATLAB language as a *DSEL* . NT^2 simplifies the development of data-parallel applications on a large selection of architectures currently including multi-core systems[29] with SIMD extensions[14]. Simply put, a MATLAB program can be converted to NT^2 by copying the original code into a C++ file and performing minor cosmetic changes (defining variables, calling functions in place of certain operators). NT^2 also takes great care to provide numerical precision as close to MATLAB as possible, ensuring that results between a MATLAB and an NT^2 code are sensibly equal.

Internally, NT^2 is designed to leverage the well known *Expression Templates* C++ idiom to build at compile time a flexible representation of the abstract syntax tree of any C++ expression containing at least one NT^2 component. This compile-time tree is then transformed in actual code to be executed on a parallel system. Contrary to other libraries based on the same technique, NT^2 relies on `Boost.Proto` , an external *Expression Templates* system to handle the creation and

transformation of ASTs [21]. `Boost.Proto` allows us to replace the direct walk-through of the compile-time AST done in most C++ *DSELs* by the execution of a mixed compile-time/runtime algorithm over the predefined AST structure generated by `Boost.Proto` (fig. 2).

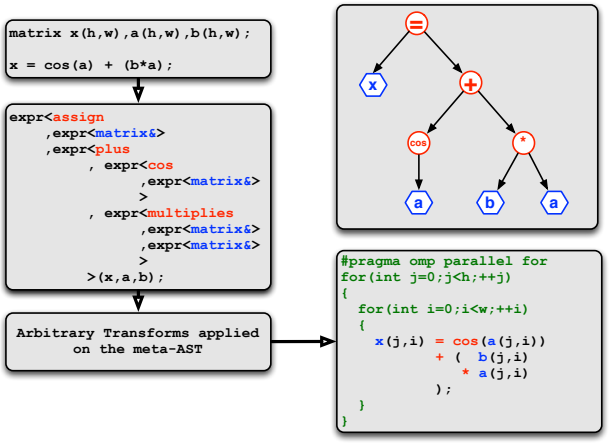


Figure 2: *Expression Templates* in NT^2

Finally, the other main difference between NT^2 and similar tools is the fact that architectural specificities of the generated code are handled by an execution model based on **Algorithmic Skeletons**[4]. Those skeletons simplify the extension of NT^2 for various hardware by separating the concerns of optimizing the ASt code for different type of architecture.

3.1 Basic NT^2 API

The main element of NT^2 is the `table` class. `table` is a template class that can be parametrized by its element type and an optional list of settings. Instances of `table` behave like MATLAB array and supports the same operators and functions. NT^2 covers a very large subset of MATLAB functionality, from standard arithmetic, exponential, hyperbolic and trigonometric functions, bitwise and boolean operations, IEEE related functions and of course linear algebra. Listing 1 showcases some NT^2 basic features including the mapping of the colon function (`:`) to the `_` object, various functions, a random number generator and some utility functions like `numel` or `size`.

```
// Matlab: A = 1:1000;
table<double> A = _(1.,1000.);

// Matlab: B = A + randn( size(A1) );
table<double> B = A + randn(size(A1));

// Matlab: r = sqrt(sum((A(:)-B(:)).^2)/numel(A));
double r = sqrt(sum(sqr(A(_)-B(_))) / numel(A));
```

Listing 1: NT^2 RMSD Computation

3.2 Support for CPU/GPU execution

If the single system computation model of NT^2 is rather classic, we needed a way to handle accelerators in a generic and extensible way. The first step is to provide a simple way to **locate** tables

on either the host system or on a device. This is simply done by using a couple of settings in the definition of NT^2 table instances. Listing 2 show cases the `nt2::host_` and `device_` settings that specifies if a `table` contains data stored on the host memory or device memory.

```
// Generates a host table by default
table<double> A( of_size(1e3,1e3) );

// Generates a host table explicitly
table<double,host_> A2( of_size(1e3,1e3) );

// Generates a device table
table<double,device_> D( of_size(1e3,1e3) );

// Generates a device table on device #2
table<double,device_> D2( of_size(1e3,1e3), on_device(2) );
```

Listing 2: NT^2 host and device specifications

Note that a special function `on_device` can be used to specify on which device the memory must be allocated in the case where multiple devices are available.

Semantic of operations between host and device tables is quite straightforward as they will be carried on the proper memory segment of each table. When mixing tables of different location, memory transfers are implicitly performed. This means that assigning a host table to a device table is equivalent to performing a CUDA memory transfer. This can be used for example to simplify interaction with existing GPU kernels as shown in listing 3. As streams are no assigned to tables, this transfer will be synchronous.

A copy function is also available to perform asynchronous memory transfers when a non-default stream is given.

```
// X is a 1e3 x 1e3 matrix full of 1.
table<double> X = ones(1e3,1e3);

// Transfer to device
table<double,device_> Y = X;

// cuBLAS direct call
cublasDscal( Y.size(), 5. , Y.data(), 1.);

// Transfer back to host
X = Y;
```

Listing 3: NT^2 interaction with cuBLAS

This semantic of transfer by assignment is a classical way of performing such operation transparently. It as been used by tools like Thrust or VexCL and have been proved to be easy enough for the user while allowing for fine grain performance tuning.

3.3 Code generation for device computation

Computations on the device occur in two situations within NT^2 : when all operations are carried out with tables set on device or if operations are carried out on any tables as long as the amount of memory to process is larger than a threshold based on the total amount of data to transfer. In those cases, the NT^2 statement applied to those tables has to be translated into an equivalent CUDA kernel and called. Contrary to some other solutions, NT^2 performs this kernel code generation **at build-time** by providing a model based on CMake macro to generate all the required calls to our kernel generation system. Once generated, the resulting `.cu` is linked with

the remaining application's binary and called directly. This choice of performing multi-stage code generation at build time was motivated by the desire to limit the overhead of runtime code generation, thus ensuring that all calls to a given kernel are efficient. The second advantage of this system is that it's fairly extensible to future situation including code generation for OpenCL devices, C code generation for embedded systems, distributed systems etc ...

Listing 4 demonstrates how high-level NT^2 code can be applied on device-enabled table.

```
table<double> X,Y,Z;
table<double,device_> A, B, C;

// full device execution
A = B + C * sin(B);

// mixed execution
X = Y / ( Z + A );
```

Listing 4: NT^2 device/host mixed code

In the first case, as all tables are defined on the device, no extra memory transfer is performed. In the second case, the mixed use of both host and device tables cause the library to starts various data stream between host and device, hence overlapping transfers and computations. If the size of the resulting table was too small, the same code would have defaulted to run in multi-core mode.

3.4 MAGMA Integration

NT^2 also provides a direct high-level integration of MAGMA [28] for handling most linear algebra tasks as shown on listing 5 for the `linsolve`.

```
table<float> A, B, X;
table<float,device_> dA, dB, dX;

X  = linsolve(A,B);
dX = linsolve(A,dB);
```

Listing 5: NT^2 interface to MAGMA kernels

Calls to `linsolve` can mix device and host tables. The implementation of such kernels will take care of transferring the strictly required data over the device and to perform transfer back to the host only if the result is assigned to a host table. Most MAGMA kernels are mapped onto ther MATLAB equivalent API and handle the same sets of optional behaviors.

3.5 Kernel optimizations

As complex programs often feature multiple statements involving NT^2 tables, it may become hard in some cases to maintain a high level of performance as the cost of transfers between statements may become significant. To locally solve this issue, NT^2 provides a function called `tie` that allows for library-based loop fusion as depicted in Listing 6.

In this scenario, the single assignment statement between call to `tie` will generate exactly one call to a single CUDA kernel composed of the fusion of the two loop nests over the variable `A` and `X`. Every table will be transferred in a single sequence of streaming operations and a single kernel will be executed on the device.

```
table<T> bs( table<T> const& B, table<T> const& C,
           , table<T> const& Y, table<T> const& Z
           )
{
  table<T> A( B.extent() ), X( Y.extent());

  tie(A,X) = tie( B + C * sin(B)
                , Y / ( Z + A )
                );

  return X;
}
```

Listing 6: NT^2 merged kernels

4 Device Code Generation

As described in section 2, code generation for accelerators in C++ is based on Meta-programming techniques. This process is however limited by the fact that C++ does not natively support language extensions for runtime code generation and program execution [26]. Creating a multi-stage paradigm is therefore necessary to remove the runtime cost of compiling CUDA code.

4.1 Multi-stage programming in C++

MSP is usually applied by adding specific language extensions to trigger a new compilation phase. As an example, MetaOcaml [27] is a multi-stage language based on Ocaml [19] with three basic constructs for runtime code optimization.

More recent adoptions of *MSP* for parallel computing include techniques like Lightweight modular staging (LMS) [24] in Scala, Language Virtualization [3] (base on LMS) or Terra [11] using language extensions for HPC in Lua . LMS is similar to Meta-programming techniques in C++ applied to the *Domain Engineering Method for Reusable Algorithmic Libraries* [6] methodology on which NT^2 is build. Language Virtualization is an interesting concept but it is limited by the need to make the *DSEL* identical to the stand-alone language. This greatly increases the complexity of the generation process and makes it hard to extend.

The way we use *MSP* is not based on language extensions but on a methodology we developed. It is possible to create a multi-stage compilation of a program using the representation described in Section 3. In C++ , we base this process on doing a compilation phase with an incomplete link to generate only an object file. It is then possible to use a demangling tool like `cppfilt` or `nm` to decode the C++ ABI names. Each demangled symbol will correspond to the internal C++ representation with a complete prototype of the incomplete function. By parsing this representation we can generate the CUDA/OpenCL kernel code with a corresponding host code to complete the link phase. Figure 3 describes this multi-stage process. To benefit from this paradigm, it is necessary to include a two stage compilation that cannot be deployed in a header only library. It is also important to develop a readable intermediate representation that can be easily parsed when demangled.

4.2 Multi-stage programming tool for NT^2

A specific tool called *symbol/code converter* (Figure 3, Part 3) coupled with a serialization process was developed as an add-on to NT^2 to solve the issues described previously. The serialization process to solve the readability of the abstract representation is based on `Boost.Serialization` . This representation is similar to the *expression-template* AST based on `Boost.Proto` that is

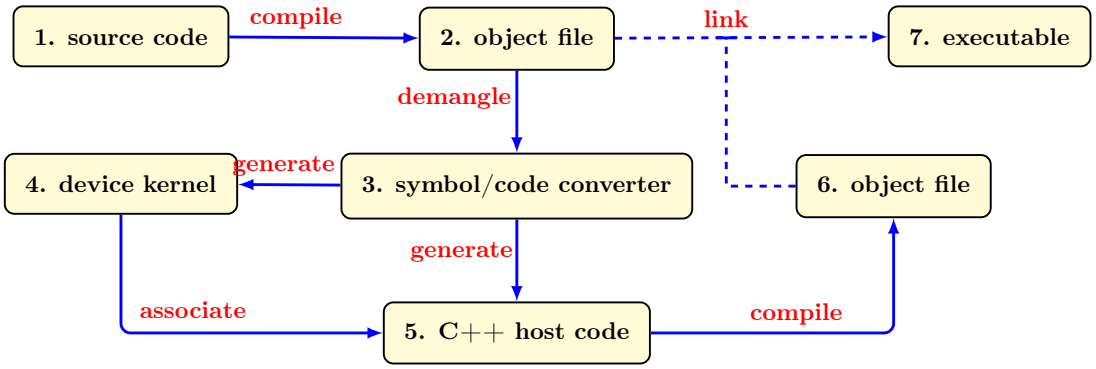


Figure 3: Two phase compilation for device code generation

parsed by the C++ compiler. This enables us to get the semantic information of the container described in section 3 like its data locality, data type or matrix shape. It is however not possible to have run-time informations like the container size.

The role of *symbol/code converter* is two-fold. First, it must parse the demangled symbols obtained. This is done with `Boost.Spirit` [9], a C++ library to parse expressions and generate outputs based on them. It is implemented as a *DSEL* using *Expression templates* and *Meta-programming* techniques. With this library, we can generate outputs corresponding to the semantic information of the containers and operators. Secondly, it must generate the device and host code (Figure 3, Part 4-5) from the output generated by `Boost.Spirit`. In order to achieve this, we must deserialize the abstract representation in which the semantic informations are represented by a tag.

We will first describe the generation of the CUDA kernel. Each representation obtained from the AST corresponds either to an elementary expression of containers such as $a = b + c$ (or $a = b + c + d \ldots$) or a sequence of operations if a fused operator (tie) is called. The parsing is separated between the left and right hand-side of the computation. The left hand-side will check if the expression is a terminal or an AST and generate the left part of the CUDA kernel expression. If it is a fused operator, it will generate a sequence of left operators in the kernel. The right hand-side will parse the operator and generate from the NT^2 CUDA back-end the corresponding operation with its parameters. Similarly, a fused operator will generate a sequence of right-hand side.

The generation of the host code consists of creating the source file corresponding to the functions with missing symbols. This amounts to adding the includes for the currents back-end, the CUDA kernel call and streaming data if necessary. As the device locality is available in the AST under the tag $nt2 :: device\_$ (see section 3), it is possible to stream the data to the GPU only if needed. Chaining non-fused operations that are on the host will obviously trigger back and forth data transfers. The overhead generated by such a process is diminished by the streaming and multi-stream process enabled for each operation.

We just described the key concepts of the *symbol/code converter* tool implemented in NT^2 for multi-stage programming. In the next section we will give concrete examples of generated kernels from *symbol/code converter* in NT^2 and how it can support hybrid computation. We will then

go on to explain the Future model for device computing.

4.3 Integration in NT^2

To describe the generation process, we use the Triad kernel which consists in doing a fused multiply-add (or fma : $a = b + c * d$). The resulting code in NT^2 is :

```
// Define host table
table<float> A,B,C,D;

// Triad kernel
 A = B + C*D;
```

Listing 7: NT^2 Triad kernel

The code in Listing 7 corresponds to Part 1 of Figure 3. During the compilation phase, the operation in Listing 7 is replaced with a call to the CUDA transform skeleton. This skeleton will then call the device kernel that is not implemented yet resulting in the incomplete link sequence(Part 1-2, Listing 7). The equivalent code for transform is detailed in Listing 8. This transform will analyze the informations on the input matrices and the architecture to decide if it should proceed with the generation process.

```
table<float> A,B,C,D;

// Triad kernel replace with transform call
 transform(A, B + C*D);
```

Listing 8: NT^2 Triad transform

From there, once we compile our code and go through phase 2 of Figure 3 we obtain the following mangled code for our transform as described in Listing 9.

```
U _ZN3nt215external_kernelINS_3tag10transform_ENS1_5cuda_
IN5boost4simd3tag4avx_EEEE4callIKNS_9container4viewINS1_6
table_EfFNS_8settingsEvEEEKNSB_10expressionINS4_5proto7exp
rns_10basic_exprINS6_4fma_ENSJ_7argsns_5list3INSC_ISD_KfSF
EESQ_SQ_EELl3EEENS_6memory9containerISD_fFSE_NS_8of_si
ze_ILln1ELln1ELln1ELln1EEEEEEEEEvRT_RT0_
```

Listing 9: NT^2 CUDA Triad mangled

We can then demangle the symbols resulting in the code described in Listing 10.

```
void nt2::external_kernel<nt2::tag::transform_, nt2::tag::cuda_<boost::simd::tag::avx_>
    >::call<nt2::container::table<float, nt2::settings()>, nt2::container::expression
    <boost::proto::exprns_::basic_expr<boost::simd::tag::fma_, boost::proto::argsns_::
    list3<nt2::container::view<nt2::tag::table_, float const, nt2::settings()>, nt2::
    container::view<nt2::tag::table_, float const, nt2::settings()>, nt2::container::
    view<nt2::tag::table_, float const, nt2::settings()> >, 3l>, nt2::memory::
    container<nt2::tag::table_, float, nt2::settings(nt2::of_size_<-1l, -1l, -1l, -1l
    )> > const>(nt2::container::table<float, nt2::settings()>&, nt2::container::
    expression<boost::proto::exprns_::basic_expr<boost::simd::tag::fma_, boost::proto::
    argsns_::list3<nt2::container::view<nt2::tag::table_, float const, nt2::settings()
    >, nt2::container::view<nt2::tag::table_, float const, nt2::settings()>, nt2::
    container::view<nt2::tag::table_, float const, nt2::settings()> >, 3l>, nt2::
    memory::container<nt2::tag::table_, float, nt2::settings(nt2::of_size_<-1l, -1l,
    -1l, -1l)> > const&)
```

Listing 10: NT^2 CUDA Triad demangled

This code corresponds to the `Boost.Proto` AST in NT^2 that we parse to generate the host and device code. The architectural tag of the machine is depicted in purple in Listing 10 and the fma computation node in red. The representation of the AST in Figure 4 corresponds to the operation obtained from the demangled symbol that we parsed.

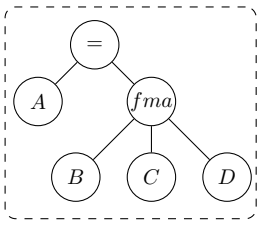


Figure 4: Triad kernel transform AST

The generated code for the .cu file is described in Listing 11 for Kepler cards. It corresponds to the AST representation with additional semantic information specific to the CUDA language. A function wrapper that calls the CUDA kernel (*fma4_ wrapper*) is used to separate the compilation of the .cu file with nvcc from the corresponding C++ host code. The triad kernel directly calls the fma function from CUDA as the NT^2 AST recognizes every occurrence of an fma and replaces it by its function (Listing 10, see boost::simd::tag::fma_). This optimization in itself is not essential since it can be done by nvcc but is still interesting as it demonstrate the potential of code generation. As NT^2 already optimizes the AST by replacing patterns with corresponding functions, we can benefit from this analysis. We can then call the corresponding CUDA function if available or our own implementation for each pattern.

```
__global__ void fma4(float* t0, const float* __restrict t1, const float* __restrict t2,
     const float* __restrict t3)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  t0[idx] = fmaf(t1[idx],t2[idx],t3[idx]);
}

void fma4_wrapper(float* t0, const float* t1 , const float* t2, const float* t3, dim3
    Grid , dim3 Block, cudaStream_t & Str, int Shr)
{
  triad<<<Grid,Block,Shr,Str>>>(t0,t1,t2,t3);
}
```

Listing 11: NT^2 CUDA Triad kernel

The host code is partially described in Listing 12 and 13. Listing 12 corresponds to the initialization of parameters and data before doing the actual computation on the device. The blockSize and stream number is determined during the generation process depending on the number of parameters and architecture. The blocksize is usually generated by measuring the bandwitdth of transfers from host to device in a range and choosing the most optimal one. To benefit best from the Kepler GPU bandwidth a block size of 40000 is necessary for single precision values. Double precision computations would lead to a lower block size generated. Similarly, the blockDim used is the high value available for the architecture as we generate element-wise operations.

```
using boost::proto::child_c;
using boost::proto::value;

std::size_t size = numel(boost::proto::child_c<0>(a1));
std::size_t blockSize = std::min(std::size_t(40000),size);
std::size_t nStreams = std::min(std::size_t(2),size/blockSize);
std::size_t n = size / blockSize;
std::size_t leftover = size % blockSize ;
dim3 blockDim = std::min(std::size_t(1024),size);
dim3 dimGrid  = blockSize/size_t(1024);
cudaStream_t stream[nStreams];

// Allocating memory on the device
```

```
value(a0).specifics().allocate(blockSize,nStreams,size,true);
value(child_c<0>(a1)).specifics().allocate(blockSize,nStreams,size);
value(child_c<1>(a1)).specifics().allocate(blockSize,nStreams,size);
value(child_c<2>(a1)).specifics().allocate(blockSize,nStreams,size);

// checks redundancy between inputs and outputs
std::unordered_set<const float*> addr;
addr.insert(child_c<0>(a0).data());

for(std::size_t i=0; i < nStreams ; ++i)
{
  cudaStreamCreate(&stream[i]);
}
```

Listing 12: NT^2 CUDA Triad Host Code 1

Depending on the number of parameters the size may be lowered to limit the allocations. The allocation process includes pinned memory and device memory allocation. If containers were defined on the GPU with $nt2::device\_$, they would not appear in the allocation phase. As we have no information on the pointer for each container, we use an unordered set to limit redundancy in memory transfers.

Listing 13 describes the computation phase. It relies on block streaming with transfers to the GPU only if NT^2 tables are on the host. This streaming process is based on the overlap data transfers concept described by NVIDIA. It consists in creating multiple streams (the number depends on the architecture and problem intensity/size) and launching for each stream a transfer host to device, the CUDA kernel and the transfers device to host for a block. As the host memory has already been allocated, we must first transfer the data to pinned memory with cudaHostAlloc to benefit from GPU optimizations. Since the difference in bandwidth between pinned and pageable memory only increases with new architectures, this optimization can give a speedup even with a mono-stream program.

```
for(std::size_t i = 0; i < n ; ++i)
{
  std::size_t j = i % nStreams;
  value(a0).specifics().transfer_htd(a0, i, stream[j], j );
  value(child_c<0>(a1)).specifics().transfer_htd(child_c<0>(a1), i, stream[j], j ,
      addr);
  value(child_c<1>(a1)).specifics().transfer_htd(child_c<1>(a1), i, stream[j], j ,
      addr);
  value(child_c<2>(a1)).specifics().transfer_htd(child_c<2>(a1), i, stream[j], j ,
      addr);

  fma4_wrapper( value(a0).specifics().data(j), value(child_c<0>(a1)).specifics().data
      (j), value(
  child_c<1>(a1)).specifics().data(j), value(child_c<2>(a1)).specifics().data(j),
      dimGrid,blockDim,stream[j]);

  boost::proto::value(a0).specifics().transfer_dth(a0, i, stream[j], j );
}

if(leftover !=0)
{
  ...
}
```

Listing 13: NT^2 CUDA Triad Host Code 2

As stated in section 3, computations on the device only occur if some conditions are met. As of now, these conditions are limited to the problem size and data locality but can be extended as the call to transform is automatic when NT^2 has defined that CUDA is available. Due to the hierarchical tag dispatching in NT^2, a system with an Intel processor coupled with an NVIDIA card will have a tag similar to the following : $cuda\_ < openmp\_ < simd\_extension >>$

. Therefore, if conditions for dispatch on the GPU are not met we will call the next level of transform (*i.e.* openmp). This enables us to use both the CPU and GPU in parallel depending on the problem which is a functionality rarely implemented in libraries. We further this process with our MAGMA back-end which does hybrid computations on most of its solvers.

The code generation is hidden from the user as the generation process is done during the standard compilation phase. The compilation overhead is negligeable as the analysis of the AST is linear and the code generation is usually not very long as seen above. The user interface only contains the added semantic information while all the complex allocations are hidden from the user.

5 Experiments

In this section, we will show that our generation process produces satisfactory performances in most situations. The benchmarks are realized with the following components :

- CPU : 2 x 6 cores Intel Xeon E5-2620 15MB L3, AVX
- GPU : Tesla K40m
  - Pageable host to device (HTD) : 3 GB/s
  - Pinned host to device : 9.8 GB/s
- Memory : 65 GB with a memcpy bandwidth of 5GB/s
- GCC 4.9, CUDA 7.0

5.1 Black & Scholes kernel

The Black & Scholes algorithm represents a mathematical model that gives a theoretical estimate of the price of European call and put options on a non-divideend-paying stock. It is a bandwidth bound algorithm for GPU if we take into account the memory transfers.

The code is given in Listing 14 using the loop-fused technique described previously with the operator tie. The $nt2::device\_$ tag is specific for accelerator enabled architectures. However, if we use the tag while no accelerator is available we will fall back to the default architecture which is $nt2::host\_$.

```
table<T> blackscholes(table<T> const& S ,table<T> const& X
                      ,table<T> const& Ta , T const r
                      , T const v
                         )
{
  auto s = extent(Ta);
  table<T , device_ > d(s) , d1(s), d2(s);
  table<T> r;

  tie(d,d1,d2,r) = tie( sqrt(Ta)
                      , log(S/X)+(fma(sqr(v),0.5f,r)*Ta)/(v*d)
                      , fma(-v,d,d1)
                      , S*normcdf(d1)-X*exp(-r*Ta)*normcdf(d2)
                      );
  return r;
}
```

Listing 14: NT^2 black and scholes

This additional semantic information on memory locality can help to avoid useless memory transfers while staying simple enough for the user.

This will result in the .cu file in Listing 15 generated for floating point values. Since the AST does not contain the name of the parameters, the kernel generator has to give a different name to each one. This does not lead to a change in performance for the kernel as we just pass multiple times the same pointer. Memory allocation on device and data transfers between host and device do pointer checking in the host code (see triad example) to insure no redundant work is done incurring also negligible overhead. The fnms function is due to an NT^2 AST transformation and corresponds to the fused negated multiply-subtract of three values.

```
__global__ void bs ( float* t0 , float* t1 , float* t2 , float* t3, const float*
    __restrict t4, const float* __restrict t5, const float* __restrict t6, const float
    t7, const float* __restrict t8, const float t9, const float* __restrict t10,
    const float t11, const float* __restrict t12, const float* __restrict t13, const
    float* __restrict t14, const float t15, const float* __restrict t16, const float*
    __restrict t17, const float* __restrict t18, const float* __restrict t19)
{
 int i = blockIdx.x*blockDim.x+threadIdx.x;
 t0[i] = sqrtf(t4[i]);
 t1[i] = plus(logf(divides(t5[i],t6[i])),divides(multiplies(t7,t8[i]),multiplies(t9,t10
     [i])));
 t2[i] = fnms(t11,t12[i],t13[i]);
 t3[i] = fnms(multiplies(t14[i],expf(multiplies(t15,t16[i]))),fastnormcdf(t17[i]),
     multiplies(t18[i],fastnormcdf(t19[i])));
}
```

Listing 15: NT^2 black and scholes fused cuda kernel

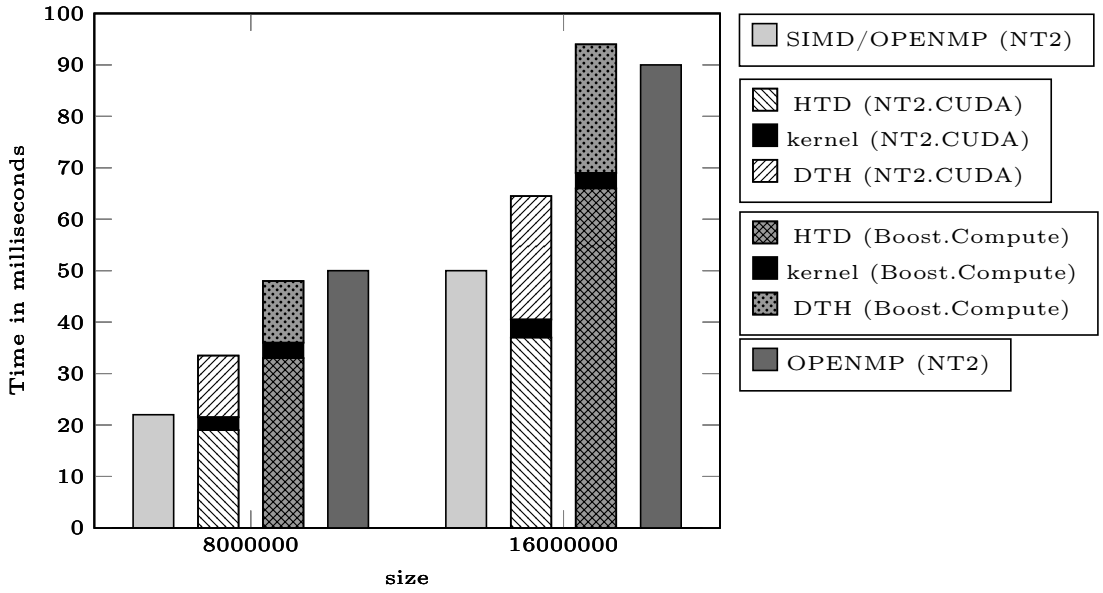


Figure 5: Black and Scholes Performance Comparison (in ms)

The Black & Scholes algorithm involves high latency and high register count operations. This will result in sub-optimal performance on SIMD for the CPU due to spilled registers while a Kepler GPU will not have any such problem. As seen in Figure 5, the execution time of the kernel on the GPU is negligible (3 ms) compared to the overall time of the optimized version with SIMD and OPENMP in NT^2 . Most of the time is spent transferring the data between

host and device memory which can be avoided with the right semantic information available. Thus, there is no overlap possible for computations. We still have better performance than the `Boost.Compute` version on which most C++ libraries are based as we use block streaming with pinned memory reaching a near-optimal throughput (average of 9.7 GB/s) on device transfers. As the bandwidth of a memcpy on the CPU (5 GB/s) is faster than page-able transfers (3GB/s) even without any overlap between transfers and computation we still increase the performance. This optimization can be disabled depending on the CUDA architecture.

As computations on the GPU are often done in great number, if the user allocates the data on the GPU he will pay no transfer cost for the rest of the computations and in this case the CUDA kernel is up to twelve times faster than the hand-optimized version for CPU.

5.2 Linsolve kernel

Linsolve is a MATLAB routine for solving linear systems. As NT^2 has its own implementation of linsolve with a LAPACK or MAGMA back-end, we can combine it with the code generation process. It also supports mixed-precision algorithms in both CPU and GPU versions through LAPACK/MAGMA or their own implementation. In this benchmark, we consider the solution of a dense linear system using the LU factorization and apply one step of iterative refinement [2]:

1. Compute $r = b - A\hat{x}$.
2. Solve $Ad = r$.
3. Update $y = \hat{x} + d$.

The equivalent NT^2 code is the following :

```
table<T, device_> A,b;
table<T, settings(device_, upper_triangular_)> r ;
table<T, settings(device_, lower_triangular_)> l ;

tie(l,u) = lu(A)
x = mtimes(trans(A),b);
x = linsolve(l,x);        // lower triangular solve
x = linsolve(r,x);        // upper triangular solve

// One-step refinement
d = b - nt2::mtimes(A,x);
d = nt2::mtimes(trans(A),d);

d = nt2::linsolve(l,d);
d = nt2::linsolve(r,d);

x = x + d;
```

Listing 16: NT^2 LU linear solve with iterative refinement

If executed on the CPU, the code in Listing 16 will call the LAPACK routines. The semantic information *upper_triangular_* allows `linsolve` to call the triangular solver instead of doing the classic linear solve. In a similar way, the `Boost.Proto` *trans*(A) node will be caught by the mtimes function and not applied but rather passed as the transpose parameter for the matrices product gemm. If executed on the GPU, the same optimizations will be applied and the iterative refinement process will trigger calls to transform for both element-wise operations.

The performance results in Figure 6 attest that the performance obtained with our model is relevant. The GPU version with *MSP* calls magma kernels using the CUBLAS dgemm routine without doing any transfer and reaches near peak performance of a K40m GPU which corresponds to 1.40 Tflop/s. The version that does not use *MSP* is slower as transfers are done

during the iterative refinement step. The CPU version quickly reaches the peak performance of both CPU which is 210 Gflop/s. As we can see, there is no performance loss while call the LAPACK/MAGMA back-ends and if device pointers are passed to our code generator, there will be no memory transfers. Similar performance would also be reached using the other factorizations available in NT^2 .

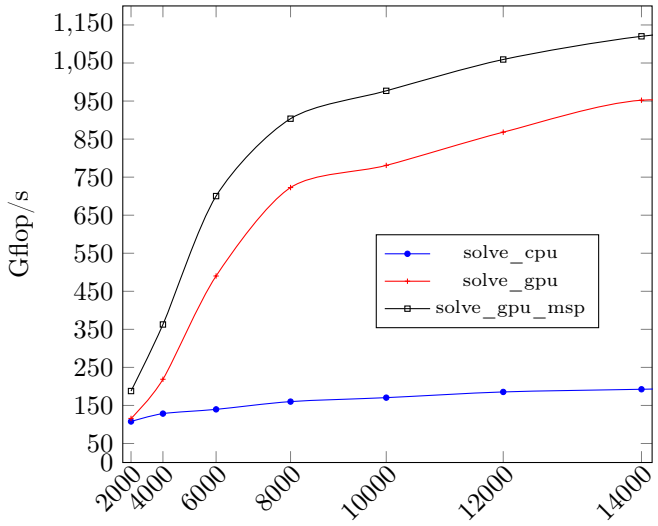


Figure 6: Performance comparison of NT^2 linear solve (in Gflop/s)

6 Conclusion

The development of tools for simplifying accelerator programming has been an active topic since accelerators have become a mainstream element of high-performance computing systems. In this paper, we proposed an extension of a high-level, data-parallel scientific computing library to specifically handle GPU accelerators. Our main objectives were to keep a similar level of expressiveness in the client code with a MATLAB -like interface while supporting different use cases of accelerator programming.

To reach this goal, we have implemented a **multi-stage** system in which the initial C++ code is used to automatically generate the equivalent CUDA kernel by reusing our internal representation of this code. This representation is based on C++ *Expression Templates* and the **Algorithmic Skeleton** to identify and classify expressions based on the kind of loop nest that is required. Finally, we showed on a selection of examples that the performance obtained is close to the hardware capability and exhibits benefits compared to other solutions.

Work is still on-going on this system, including the final integration into the main NT^2 release and support for more specific functions on the latest GPUs. Implementing a more thorough cost model to ensure better scheduling of computation between CPU and GPU is also being studied. The natural evolution of this work is to extend our approach to the Intel Xeon Phi coprocessor, the runtime-based CUDA compiler recently released, or OpenCL devices. An interesting track of research can be derived from the support of OpenCL by targeting OpenCL enabled FPGAs, as NT^2 could bridge between high-level C++ and hardware design.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[3] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010.

[4] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[5] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012.

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.

[7] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative Programming and Active Libraries. In *International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, pages 25–39, 1998.

[8] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, pages 51–72, 2003.

[9] Joel de Guzman. The boost spirit parser generator framework, 2008. *URL http://spirit. sourceforge. net*.

[10] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. Programming CUDA and OpenCL: a case study using modern C++ libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472, 2013.

[11] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

[12] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. *New Gen. Comput.*, 25(3):305–336, January 2007.

[13] Pierre Esterie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté, and Lionel Lacassagne. The numerical template toolbox: A modern c++ design for scientific computing. *Journal of Parallel and Distributed Computing*, 74(12):3240–3253, 2014.

[14] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost.SIMD: generic programming for portable SIMDization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012.

[15] Gael Guennebaud, Benoit Jacob, et al. Eigen: A C++ Linear Algebra Library. `http://eigen.tuxfamily.org/`, 2014.

[16] Jared Hoberock and Nathan Bell. Thrust: A parallel template library. *Online at http://thrust. googlecode. com*, 42:43, 2010.

[17] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[18] Stephan C Kramer and Johannes Hagemann. SciPAL: Expression templates and composition closure objects for high performance computational physics with cuda and openmp. *ACM Transactions on Parallel Computing*, 1(2):15, 2015.

[19] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.12. *Documentation and user's manual. INRIA*, 2010.

[20] Kyle Lutz. Boost.Compute. `http://github.com/kylelutz/compute`, 2015.

[21] Eric Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007.

[22] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[24] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[25] Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. *Proc. GPUScA*, pages 51–56, 2010.

[26] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[27] Walid Taha, C Calcagno, X Leroy, E Pizzi, E Pasalic, JL Eckhardt, R Kaiabachev, O Kiselyov, et al. Metaocaml-a compiled, type-safe, multi-stage programming language, 2006. *See: http://www. metaocaml. org*.

[28] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010.

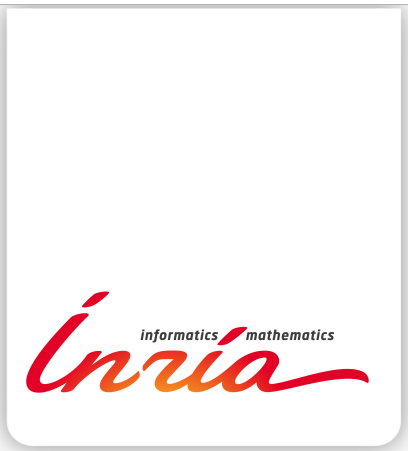
[29] Antoine Tran Tan, Joel Falcou, Daniel Etiemble, and Hartmut Kaiser. Automatic task-based code generation for high performance domain specific embedded language. *7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)*, 2014.

[30] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[31] Todd L. Veldhuizen and Dennis Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98*. SIAM Press, 1998.

[32] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

Contents



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399